# Habib University
# Computational Intelligence - CS 451

## Assignment 01 - Report
## Evolutionary Algorithms

**Instructor:** Dr. Saleha Raza

Ali Muhammad Asad - aa07190
Muhammad Yousuf Uyghur - mu07486

# Contents

# 1 Question 1 - Evolutionary Algorithms

## 1.1 Genetic Algorithm Implementation

For the genetic algorithm, a separate class called `EvolutionaryAlgorithm` was implemented, that has attributes and methods suited for our evolutionary step.

### 1.1.1 Parameters

The given parameters in the pdf file were tweaked and tested upon, and eventually, the following default parameters were chosen to run our tests on as they gave better results:

- Population Size: 100
- Offspring Generation: 70
- Number of Generations: 1000
- Mutation Rate: 0.95
- Number of Iterations 10

Having a larger population size includes more diversity in our population as a larger number of chromosomes are initialized. Similarly, having a higher offpring generation means more variety of offsprings can be generated, and thus chosen to go in the next generation. The offspring count has been kept lower than the population size so as to not take too much time. The generations have been kept at 1000 to have a better reading where the values converge. Finally, having a higher mutation rate also contributes to diversity in our population, so that our problem has a lesser likelihood of being stuck at a sub-optimal solution.

### 1.1.2 Parent Selection Implementation

The `ParentSelection` method takes as argument a `selection` scheme in the form of a string, and that selection scheme is then used to select the parents that are used to create offsprings through crossover in between the two parents, followed by mutation on the selected offsprings based on the mutation rate. The code is as follows:

```python
def ParentSelection(selection: str) -> None:
    selection_method = getattr(SelectionSchemes, selection)
    parents = selection_method(population, fitness_vals, Num_Offsprings)
    for i in range(0, Num_Offsprings, 2):
        child1 = crossover(parents[i], parents[i + 1])
        child2 = crossover(parents[i], parents[i + 1])
        population.append(mutate(Mutation_Rate, child1))
        population.append(mutate(Mutation_Rate, child2))
```

Listing 1: Parent Selection Algorithm

### 1.1.3 Survival Selection Implementation

The `SurvivalSelection` method takes as argument a `selection` scheme in the form of a string, which is then used to select the individuals that will be sent on to the next generation, based on the selection scheme. Thus, once the new offsprings have been added, this method reduces the population size back to the default setting. In my implementation, we have followed an elitism approach, which ensures that the best individual always survives to the next generation.

```python
def SurvivalSelection(selection: str) -> None:
    problem.population = population
    fitness_vals = problem.fitness()
    selection_method = getattr(SelectionSchemes, selection)
    new_gen = selection_method(population, fitness_vals, Population_Size
    - 1)
    best_individual = getBestIndividual()
    new_gen.append(best_individual)
    population = new_gen
    problem.population = population
    fitness_vals = problem.fitness()
    generation = generation + 1
```

Listing 2: Survival Selection Algorithm

### 1.1.4 Local Search and Replace

Additionally, another method was defined and used, `LocalSearchAndReplace`, which performs a local search around the best individual in the population. It does this by creating a mutated version of the best individual and evaluating its fitness. If the mutated individual has a higher fitness than the current best, or if its fitness is greater than or equal to the average fitness of the population, it replaces the worst individual in the population. This method can help to improve the overall fitness of the population and potentially escape from local optima. The method works as follows:

```python
def LocalSearchAndReplace() -> None:
    best_individual = getBestIndividual()
    mutated_best = self.problem.mutate(1.1, best_individual)
    mutated_fitness = self.problem.fitness_helper(mutated_best)
    if mutated_fitness < getBestFitnessScore() or mutated_fitness <
    getAvgFitnessScore():
        worst = fitness_vals.index(getWorstFitnessScore())
        population[worst] = mutated_best
        fitness_vals[worst] = mutated_fitness
```

Listing 3: Local Search and Replace Implementation

## 1.2 Traveling Salesman Problem - TSP

In my TSP Implementation, a single chromosome represents a tour that the salesman takes. It's a list of city indices, where each index represents a city. The order of indices in the list represents the order in which the cities are visited in the tour. Additonally, since each city has some coordinates, between any two cities there has to be some distance. Therefore, the fitness function simply takes a chromosome, and for every two cities in order, it finds the distance in between and adds them to a total distance variable, thus giving the total fitness of the chromosome. Since TSP is a minimizing problem, the lower the fitness score, the better the chromosome.

#### Initialize Population

This method generates the initial population of tours. I first experimented with a completely random generation of tours, however, that resulted in a very high initial value of tours, and took a high number of generations to reach optimal values. Therefore, a slightly different heuristic was taken for initializing the population. For each tour in the population:

- With a 90% probability, it starts with a randomly chosen city and then adds the nearest unvisited city to the tour. Therefore, its a little bit greedy, but also initializes the population with potentially good chromosomes, due to which, we start at better fitness values from the get-go.
- With a 10% probability, it generates a random tour by shuffling the list of all cities.

### Mutation

The mutation method, based on the mutation rate, randomly selects two points, p1 and p2, which serves as the bases for which we split our chromosome into three parts; first, middle, and end. Then the middle part is inserted back into the chromosome, but in reversed order.

```python
def mutate(Mutation_Rate: float, Chromosome: list) -> list:
if random.random() < Mutation_Rate:
    p1 = random.randint(0, len(Chromosome)-1)
    p2 = random.randint(p1, len(Chromosome)-1)
    chrom_middle = Chromosome[p1:p2]; chrom_first = Chromosome[:p1];
    chrom_end = Chromosome[p2:]
    Chromosome = chrom_first + chrom_middle[::-1] + chrom_end
return Chromosome
```

Listing 4: Mutation Algorithm

### Crossover

This method performs a crossover operation between two parent tours to generate a child tour. It randomly selects a segment from the first parent and keeps it in the child. It then fills the rest of the child with the genes from the second parent in their original order, skipping those genes that are already included in the child from the first parent. If a gene from the second parent is already included in the child, it replaces it with the corresponding gene from the first parent. If there are still duplicate genes in the child, it replaces them with the missing genes.

```python
def crossover(parent1: list, parent2: list) -> list:
    pivot1 = random.randint(0, len(parent1)-1)
    pivot2 = random.randint(pivot1, len(parent1)-1)
    middle = parent1[pivot1:pivot2]
    first = parent2[0:pivot1]; end = parent2[pivot2:len(parent2)]
    for i in range(len(end)):
        if end[i] in middle: end[i] = parent1[pivot2 + i]
    for i in range(len(first) -1, -1, -1):
        if first[i] in middle and first[i] not in end: first[i] = parent1
    [i - pivot1]
    child = first + middle + end
    for i in range(len(child)):
        if child.count(child[i]) > 1:
            if parent1[i] not in child: child[i] = parent1[i]
            elif parent2[i] not in child: child[i] = parent2[i]
            else: child[i] = [x for x in parent1 if x not in child][0]
    return child
```

Listing 5: Crossover Algorithm

## 1.3 Job Shop Scheduling Problem - JSSP

In my JSSP Implementation, a single chromosome represents a sequence of operation, where each operation is a tuple of a job and a machine. The order of operations in the chromosome represents the order in which the operations are processed. The population is initialized by creating random permutations of the operations. The fitness function initliazes two dictionaries, `machineEndTimes`, and `jobEndTimes` to keep track of when a machine and job finishes its current operation. It then iterates over each operation in the chromosome. For each operation, it finds the start time, which is the maximum of the end times of the current job and the current machine. This ensures that a machine can only start a new operation after it has finished its current operation, and a job can only start a new operation on a machine after it has finished its current operation.

The end time of an operation is the start time plus the processing time. The function updates machineEndTimes and jobEndTimes with the end time.

Finally, the function returns the maximum end time across all machines, which represents the total time it takes to complete all jobs. Since we want to minimize this time.

Since JSSP is an optimizing problem where we want to reduce the time taken, the lower the fitness score, the better the chromosome.

### Mutation

The mutation method used in the implementation is an insert mutation. It randomly selects two positions in the chromosome, removes the operation at the first position, and inserts it at the second position. The mutation is performed with the provided mutation rate.

```
def mutate(Mutation_Rate: float, Chromosome: list) -> list:
    if random.random() < Mutation_Rate:
        p1 = random.randint(0, len(Chromosome)-1)
        p2 = random.randint(0, len(Chromosome)-1)
        while p1 == p2:
            p2 = random.randint(0, len(Chromosome)-1)
        operation = Chromosome.pop(p1)
        Chromosome.insert(p2, operation)
    return chromosome
```

Listing 6: Mutation Algorithm

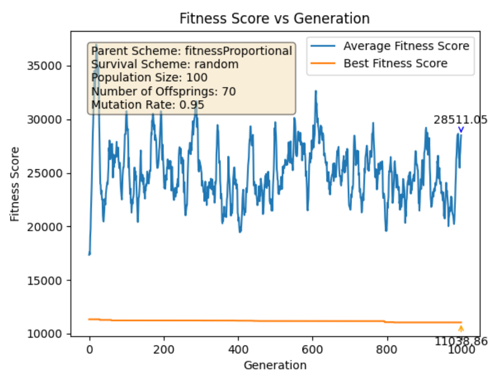
### Crossover

The crossover method used in the implementation is a variant of order crossover. It randomly selects a segment from one parent and fills in the rest of the genes from the other parent in the order they appear, while ensuring that each operation appears exactly once in the child chromosome. The crossover is directly implemented using the same logic as it was for TSP above. You can refer to that pseudocode for the crossover which can also be found here.

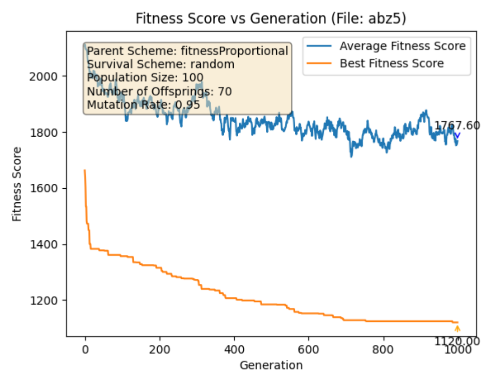
## 2 Statistics and Analysis - Evolutionary Algorithms (Q1)

The below generated plots each follow the same parameters as mentioned above in the start of the pdf in the Parameters section. In each of the scehemes, the first scheme mentioned is the Parent Scheme, and the second scheme is the Survival Selection Scheme.

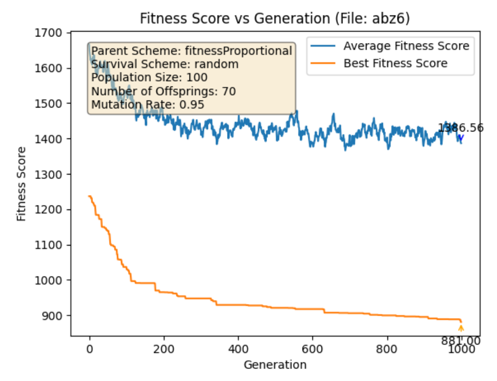
### 2.1 Fitness Proportion Scheme (FPS) and Random



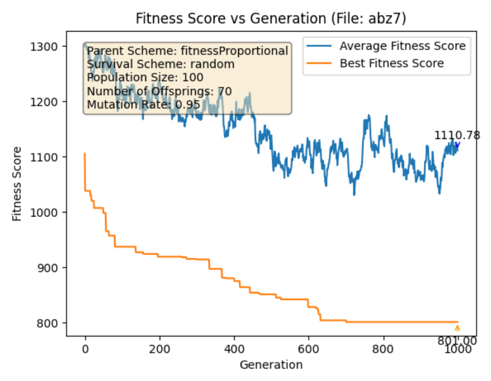
(a) TSP



(b) JSSP - abz5



(c) JSSP - abz6

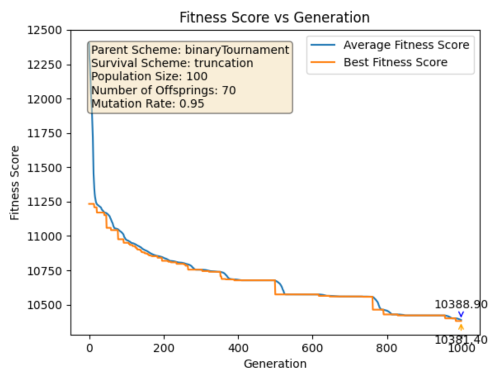


(d) JSSP - abz7

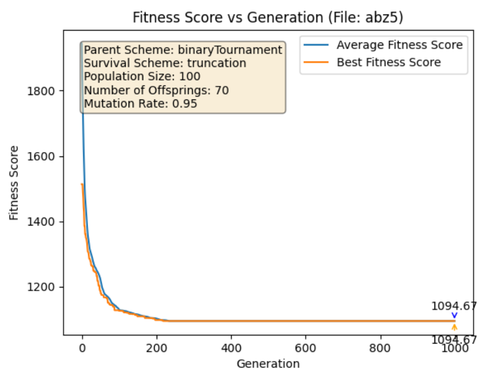
Figure 1: Plots with FPS and Random schemes

As shown above, although we get a good value for the best chromosome in the population, but due to the random nature of our selection scheme, our average has a lot of variation and doesn't seem to converge towards a value, since it keeps a high diversity. Therefore, this is not a good scheme combination.

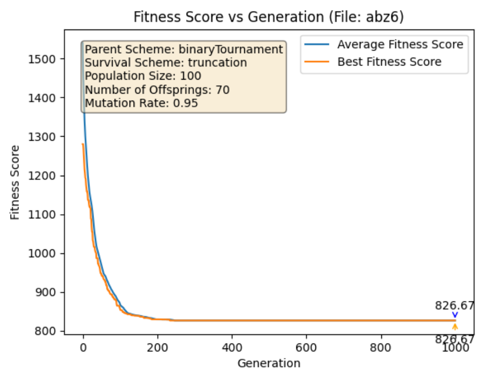
## 2.2 Binary Tournament and Truncation



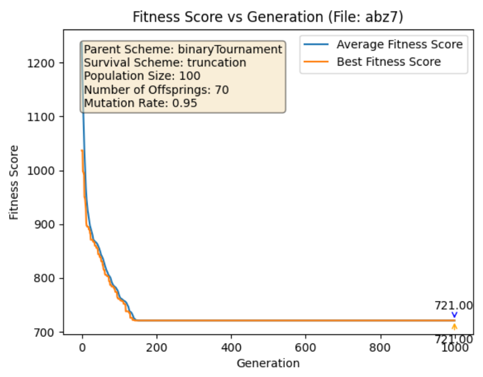
(a) TSP



(b) JSSP - azbz5



(c) JSSP - abz6

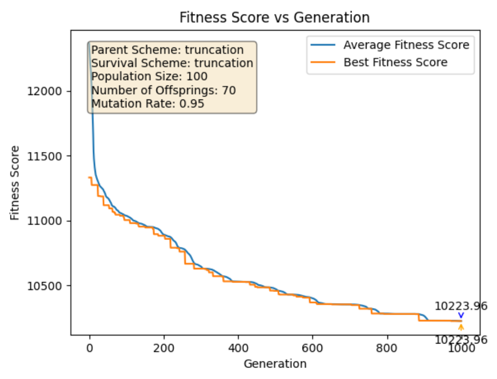


(d) JSSP - abz7

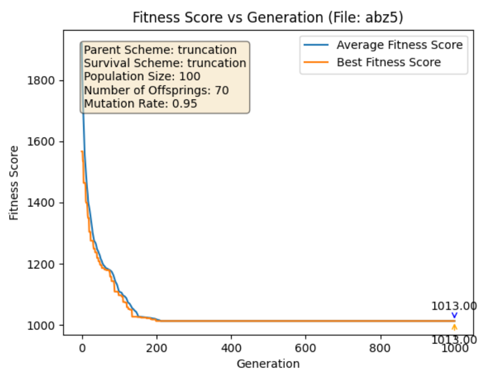
Figure 2: Plots with Binary Tournament and Truncation schemes

In the Binary Tournament Selection Scheme, the default tournament size has been kept as the Population Size in order to make implementation easier for us. This is a clear improvement over the previous combination scheme, as we can see the values are not only improving, but also converging towards the best value. The binary parent scheme ensures that the best parents are selected, and the truncation survival scheme ensures that only the fittest candidates move on to the next generation. This coupled with a high mutation rate ensures that there is a good balance of exploration, and exploitation in our population by including diversity as well, hence we get good values.

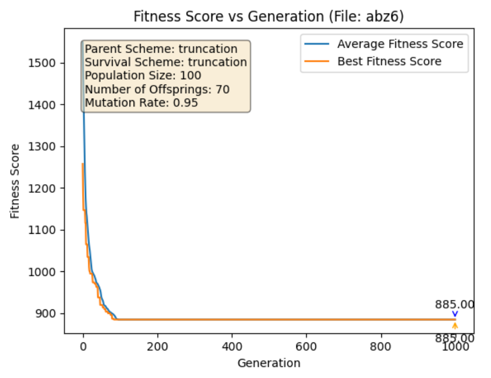
## 2.3 Truncation and Truncation



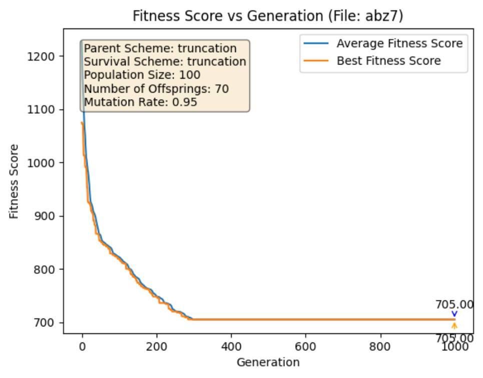
(a) TSP



(b) JSSP - azbz5



(c) JSSP - abz6

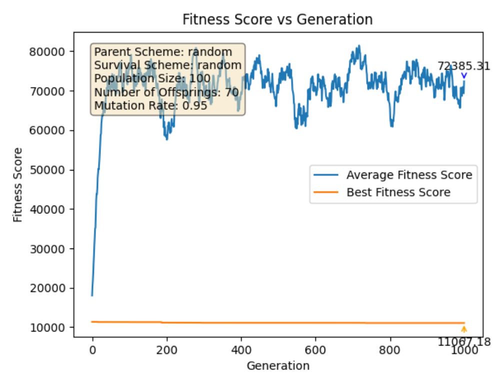


(d) JSSP - abz7

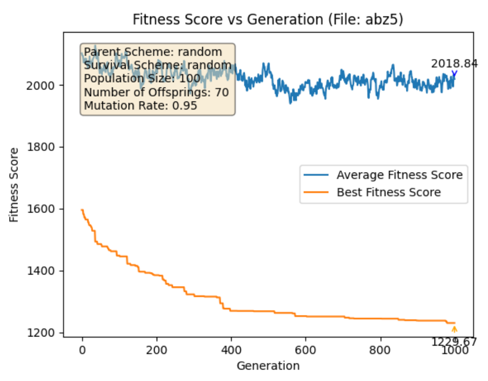
Figure 3: Plots with Truncation scheme

In this combination, since both schemes are truncation, then only the best of the chromosomes are selected as parents, and only the best of the chromosomes are selected for the next generation. This tends to be more exploitative, which can also mean that the population can quickly converge to a sub-optimal solution due to lack of diversity, therefore, a high value of exploration should be there. Our high mutation rate, and `LocalSearchAndReplace()` methods take care of this, providing a good counter to the high exploitativeness of this combination. Thus we get better results, and don't always get stuck at sub-optimal points either.

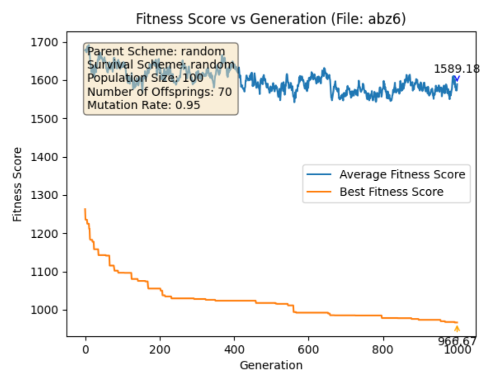
## 2.4 Random and Random



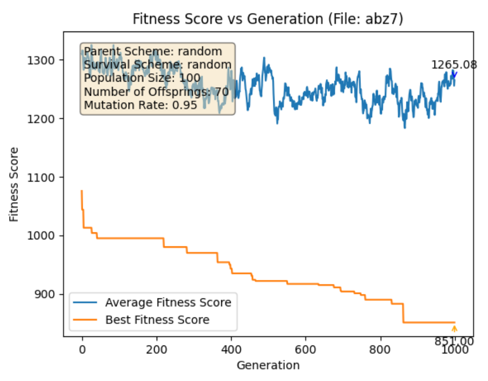
(a) TSP



(b) JSSP - azbz5



(c) JSSP - abz6

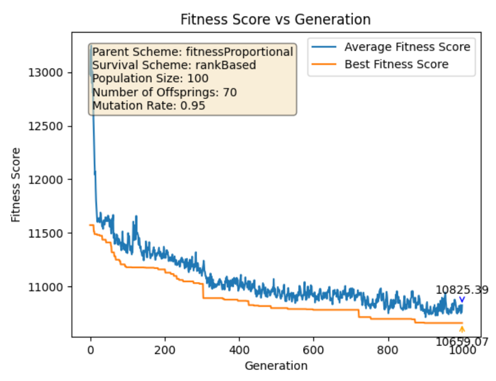


(d) JSSP - abz7

Figure 4: Plots with Random scheme

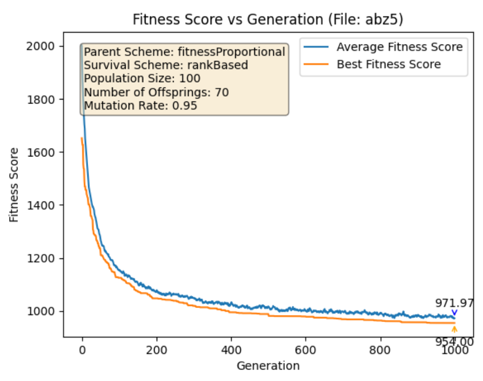
Probably the worst combination we can take, since everything is completely random, therefore, there doesn't seem to be any apparent convergence, or improvement. Rather, at some occasions our population has also gotten worse on average.

## 2.5 Fitness Proportion Scheme (FPS) and Rank Based Scheme (RBS)



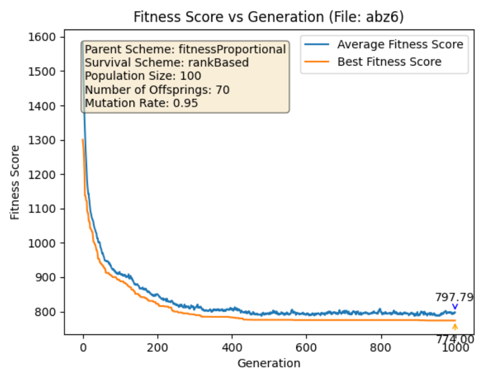
Parent Scheme: fitnessProportional
Survival Scheme: rankBased
Population Size: 100
Number of Offsprings: 70
Mutation Rate: 0.95

(a) TSP



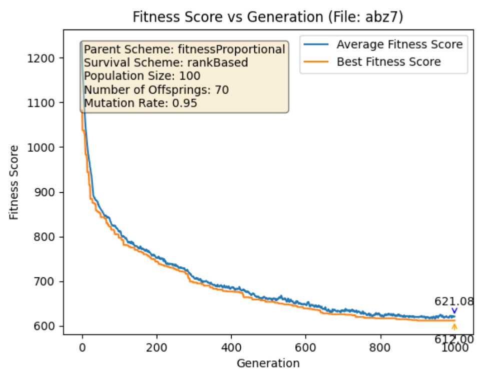
Parent Scheme: fitnessProportional
Survival Scheme: rankBased
Population Size: 100
Number of Offsprings: 70
Mutation Rate: 0.95

(b) JSSP - azbz5



Parent Scheme: fitnessProportional
Survival Scheme: rankBased
Population Size: 100
Number of Offsprings: 70
Mutation Rate: 0.95

(c) JSSP - abz6



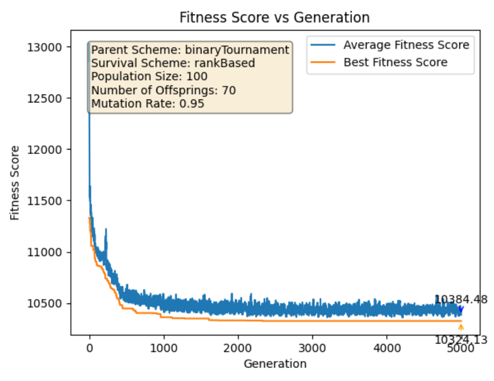
Parent Scheme: fitnessProportional
Survival Scheme: rankBased
Population Size: 100
Number of Offsprings: 70
Mutation Rate: 0.95

(d) JSSP - abz7

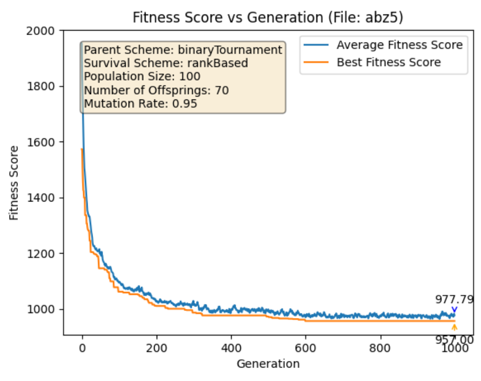
Figure 5: Plots with FPS and Rank Based schemes

Since our parents are selected proportional to their fitness, and the survivors are selected according to ranks, we see a good balance between exploration and exploitation. This also preserves diversity in our population as we can see in the varying average fitness, while also seeking better values.

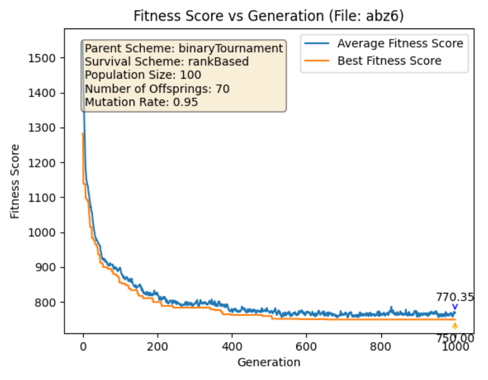
## 2.6 Binary Tournament and Rank Based



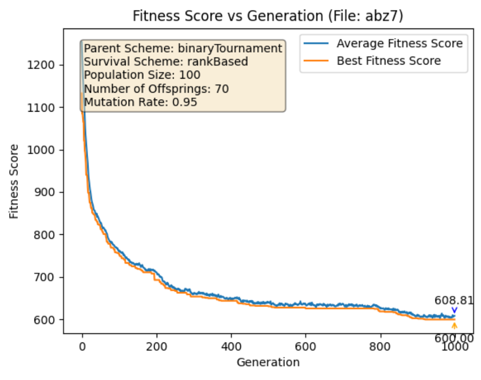
(a) TSP



(b) JSSP - azbz5



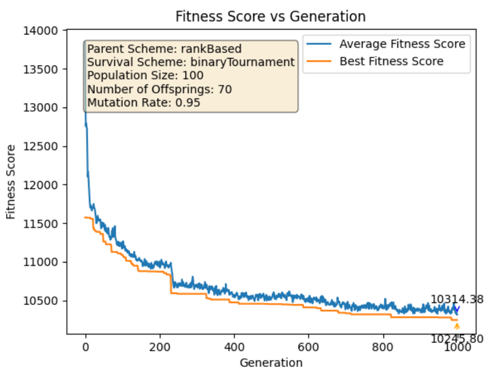
(c) JSSP - abz6



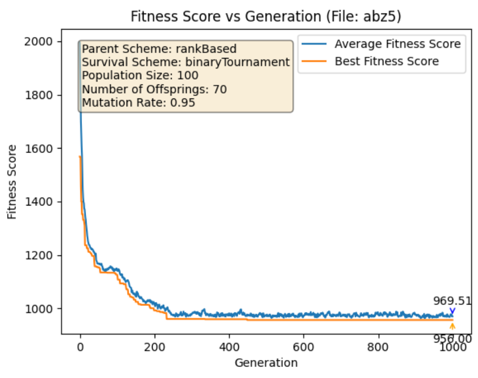
(d) JSSP - abz7

Figure 6: Plots with Binary Tournament and Rank Based

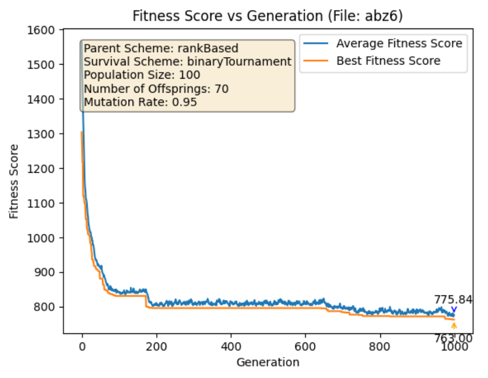
## 2.7 Rank Based and Binary Tournament



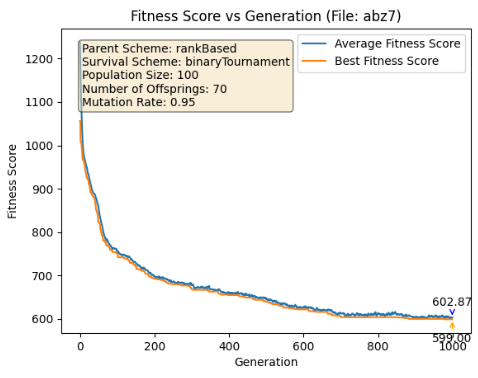
(a) TSP



(b) JSSP - azbz5



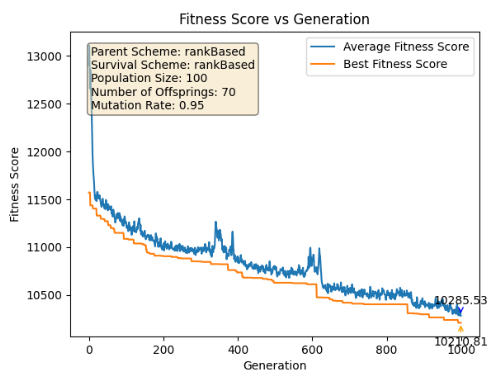
(c) JSSP - abz6



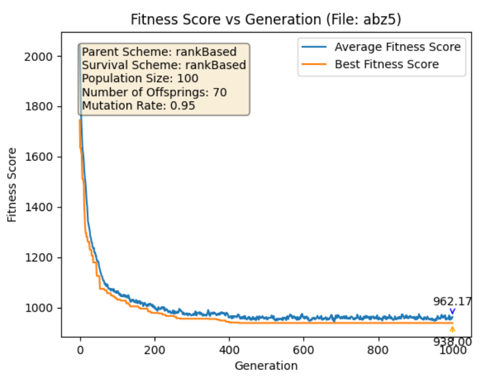
(d) JSSP - abz7

Figure 7: Plots for Rank Based and Binary Tournament Schemes

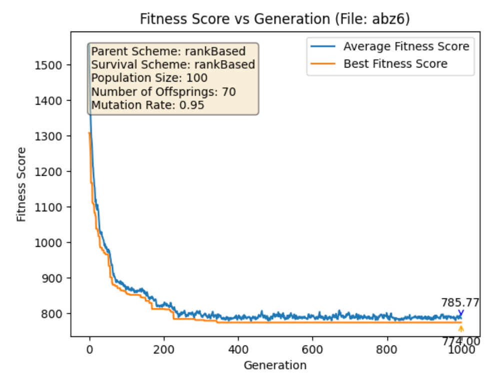
## 2.8 Rank Based and Rank Based



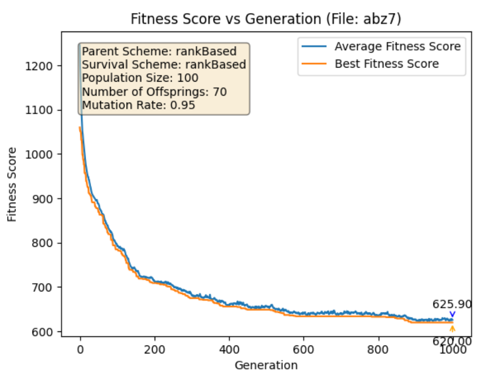
(a) TSP



(b) JSSP - azbz5



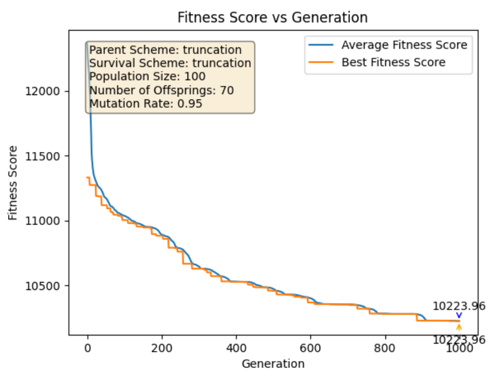
(c) JSSP - abz6



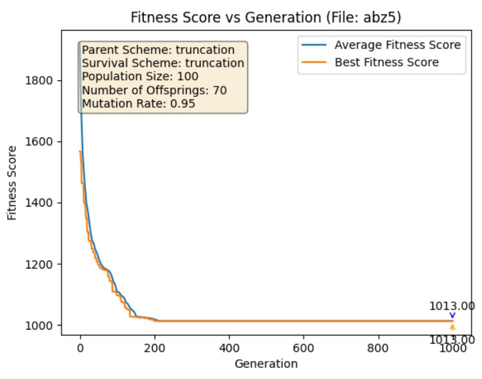
(d) JSSP - abz7

Figure 8: Plots for Rank Based schemes

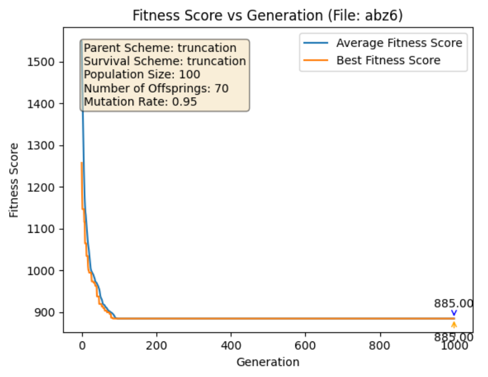
## 2.9 Truncation and Truncation



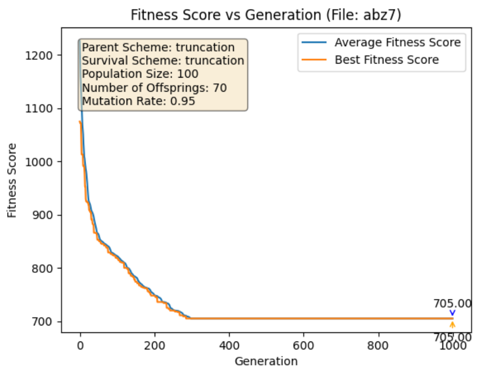
(a) TSP



(b) JSSP - azbz5



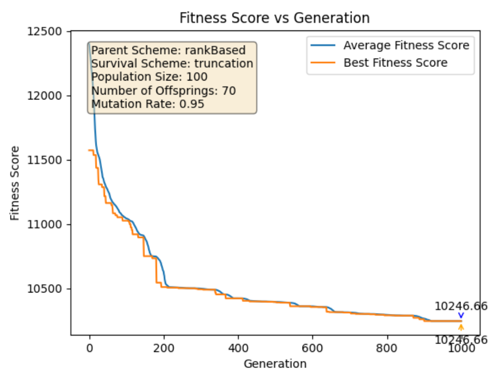
(c) JSSP - abz6



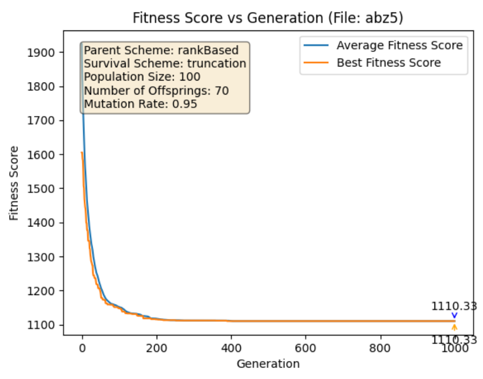
(d) JSSP - abz7

Figure 9: Plots for Truncation schemes

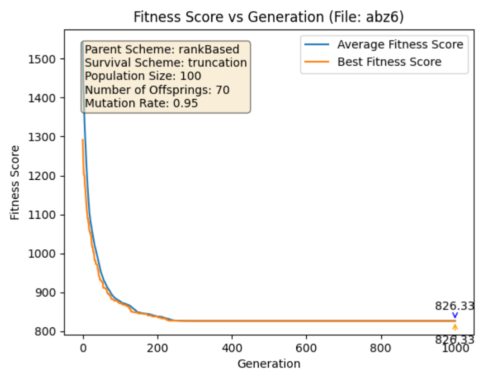
## 2.10 Rank Based and Truncation



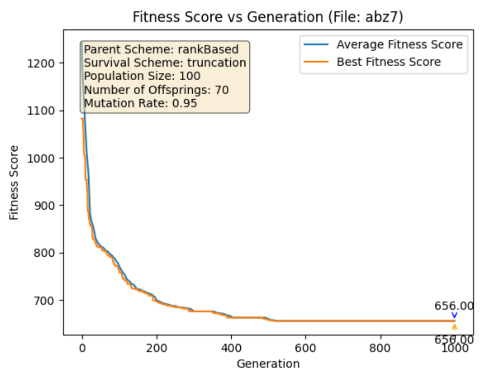
(a) TSP



(b) JSSP - azbz5



(c) JSSP - abz6



(d) JSSP - abz7

Figure 10: Plots for Rank Based and Truncation Schemes

## 2.11 Analysis and Personal Best Values

In those combinations where we have “Fitness Proportion Scheme”, and “Random Scheme”, we noticed a really high variation in our average values. Fitness still performs better, but Random introduces very high variations in our values, and hinders convergence and improvement. Therefore, after some combinations, it was decided to leave combinations involving these schemes since we already saw better results in the other combinations.

In combinations involving “Binary Tournament”, “Rank Based”, and “Truncation”, we observe a lot of improvement, and convergence in our population. This is particularly true with the case of truncation, which might be due to its inherent nature of selecting those chromosomes with better fitness values. In Binary and Rank Based combinations, we can still see some variations in our average fitness, although, overall, it seems to decrease and follow the best fitness trendline, which shows that not only the best individuals are selected in each evolutionary cycle, since binary and rank based both offer diversity as they have a chance to pick other individuals apart from the best one. Due to this, they also provide a higher exploration mechanism, and less likelihood of being stuck at a sub-optimal solution. For JSSP, we see that schemes involving rank and binary tournament mostly give the best results. So does for TSP.

## 2.12 Best Results for TSP

The following images show the best results I got for TSP, on various schemes and parameters.

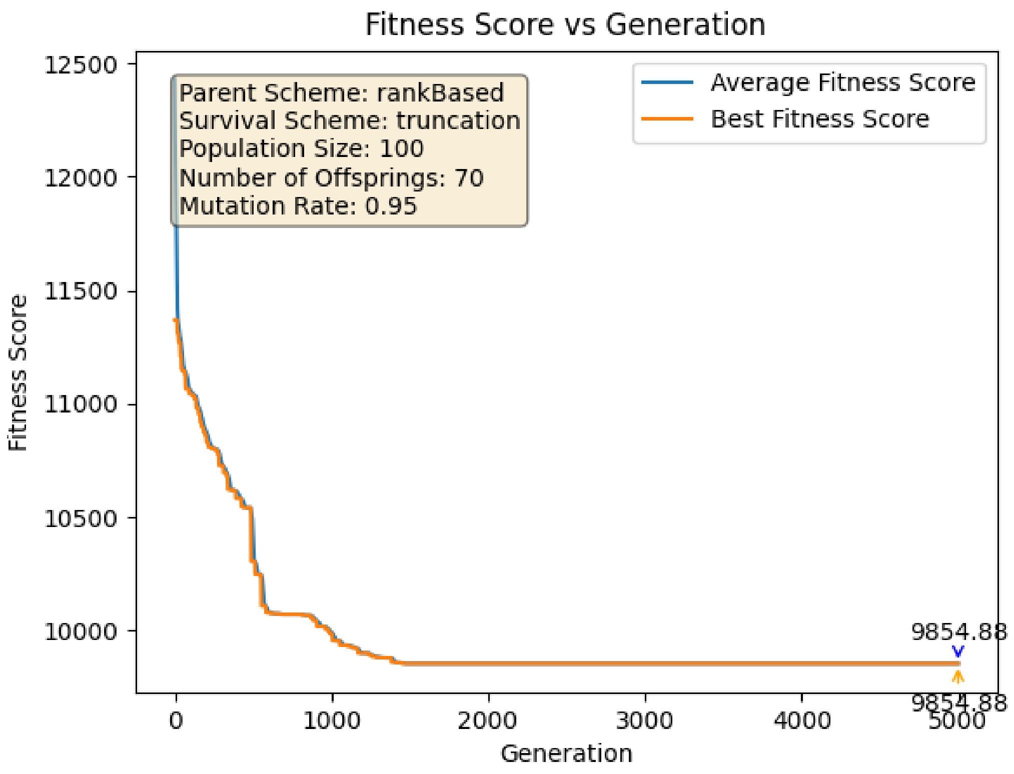


Figure 11: Best Overall TSP Score

The above image shows the best score I received for the TSP problem. For this, the following settings and parameters were used:

- Parent Scheme: Rank Based
- Survival Scheme: Truncation
- Population Size: 100
- Offsprings: 70
- Mutation Rate: 0.95
- Initialization of Population was done greedily
- Elitism was used in the survivor selection algorithm
- Local Search and Replace was Implemented
- I believe I got lucky with this run as well

The following images are also some of the best results I got, with somewhat similar settings.

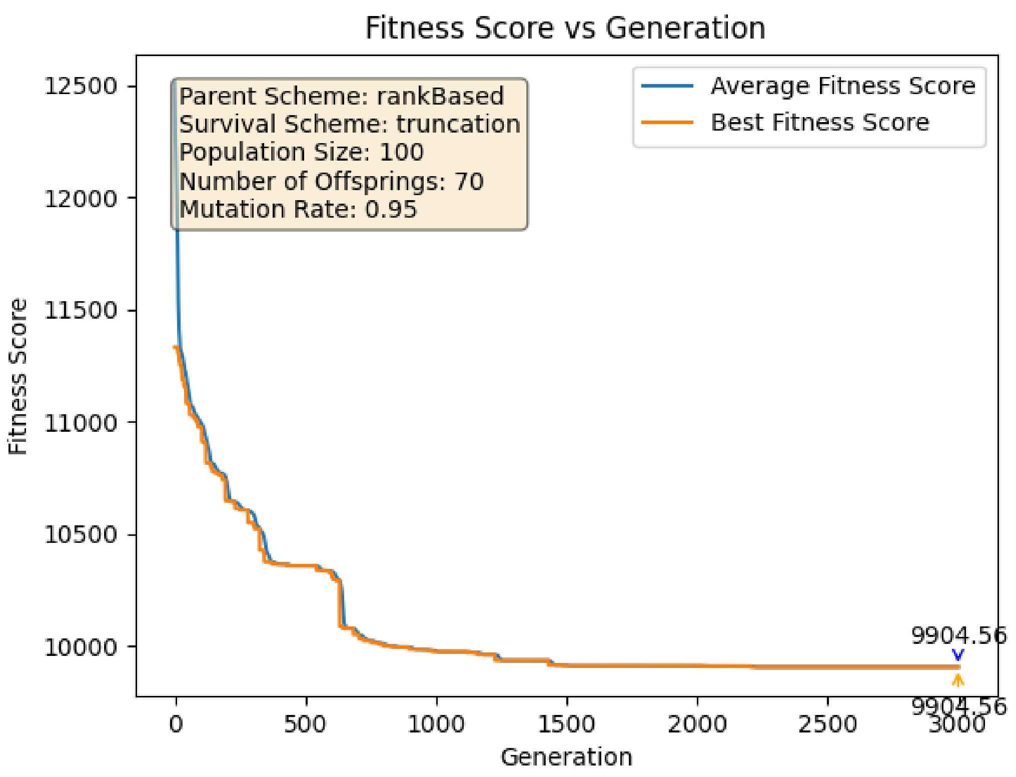

Parent Scheme: rankBased
Survival Scheme: truncation
Population Size: 100
Number of Offsprings: 70
Mutation Rate: 0.95

Figure 12: Second Best Result

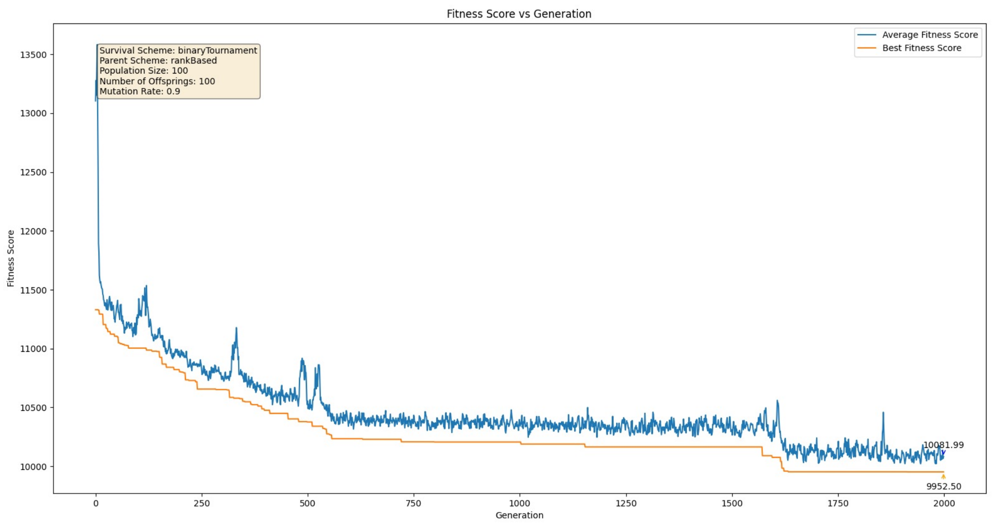

Survival Scheme: binaryTournament
Parent Scheme: rankBased
Population Size: 100
Number of Offsprings: 100
Mutation Rate: 0.9

Figure 13: Third Best Result - using Rank Based and Binary Tournament

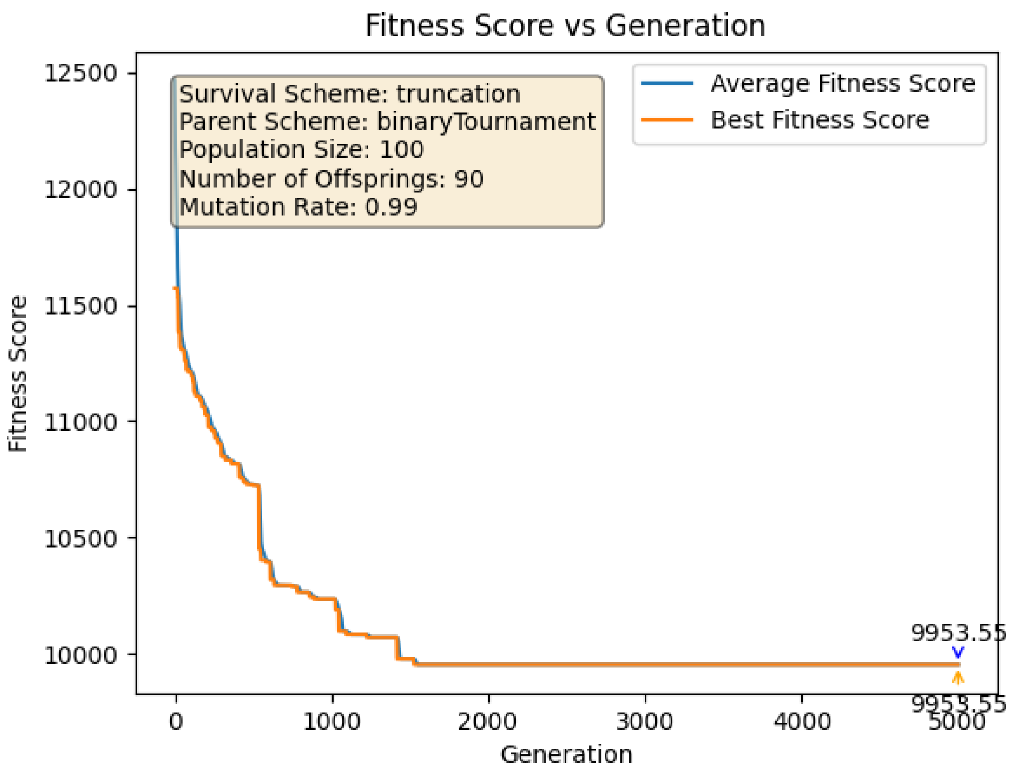


Figure 14: Fourth Best Result - using Binary Tournament and Truncation

## 3 Evolutionary Art